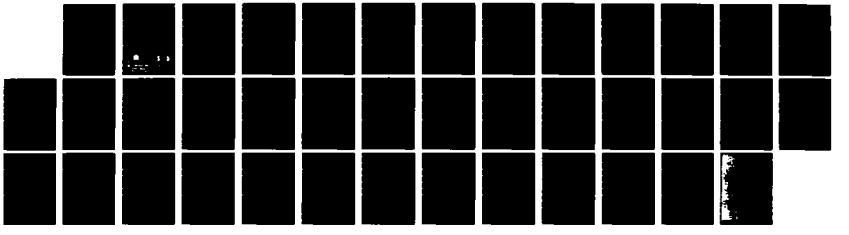
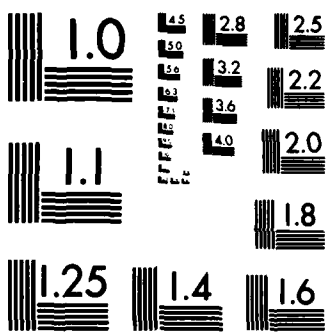
AD-A127 572 A CASE STUDY IN WRITING EFFICIENT PROGRAMS(U) 1/1
CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER
SCIENCE J L BENTLEY 25 JAN 83 CMU-CS-83-108
UNCLASSIFIED N00014-76-C-0370 F/G 9/2 NL





CMU-CS-83-108

15

A Case Study in Writing Efficient Programs

Jon Louis Bentley'
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

30 January 1983

Carnegie-Mellon University

CMU-CS-83-108

A Case Study in Writing Efficient Programs

Jon Louis Bentley[1]
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

25 January 1983

Abstract -- The time and space performance of a computer program rarely matter, but when they do they can be of crucial importance to the success of the overall system. This paper discusses the performance issues that arose in the implementation of a small system (twelve programs, two thousand lines of code) for a small business. The paper emphasizes a general methodology for improving system performance; the details of the case study show how the general techniques are applied in a particular context.

[1]The system development described in this paper was performed as consulting; the research involved in extracting and describing the general principles was supported in part by the Office of Naval Research under Contract N00014-76-C0370.

Table of Contents

1. Introduction

It takes a lot to make a computer system useful: it has to do what the user wants it to do, it has to be comfortable for the user to use, and it has to be maintainable so the programmer can fix bugs and make enhancements. It is a tragic fact of programming life that a program that is perfect in ninety-nine out of a hundred categories but fails in that hundredth can be useless -- just like an almost-perfect racing car in which the designers forgot the steering wheel.

This paper tells the story of a computer system that was good enough in all ways but one -- it was too slow. I thought that one task to be run several times a day would take just a few minutes; it actually took *fourteen hours*. Fortunately, there are two pieces of good news from this story:

- I was able to reduce the run times of the expensive parts of the system enough to make the system useful. The microcomputer-based system successfully replaced a company's punched-card, mainframe-based system, and performs a broader task more cheaply using less operator time.

- The speedups were achieved not by wizardry known only to expert sorcerers, but by applying some simple engineering techniques.

The purpose of this paper is to tell the story of how I made the system more efficient and to enumerate the engineering techniques underlying the story. Although the story deals with a medium-sized system on a microprocessor, the techniques apply to programs from a few dozen lines of code on a microprocessor to mainframe systems of hundreds of thousands of lines.

Before I go into more background of story, I should give a few more technical details on the system. It was written primarily in Disk BASIC on a Radio Shack TRS-80 Model III computer; it consisted of about a dozen programs for a total of 2000 lines of code. Some specific speedups I'll describe later include the following.

- The code to be run several times a day that took fourteen hours was reduced to two hours and twenty minutes by fine-tuning. The system was also reorganized so that the code was used only about one-third as frequently.

- A data entry program was originally so slow that it lost characters when the operator typed too fast; it was reorganized to keep up with the fastest operators. In its original implementation it sometimes took forty-five seconds to process a record after it was completely typed; the final program could do the same job in about one second.

- The run time of a data processing program was reduced from fifteen hours to five; the run

time of a text processing program was reduced from fifteen minutes to seven; the run time of a graphics program was reduced from eight minutes to three and a half minutes.

Those facts appeal to programmers; let me also include a couple of facts that were of more interest to the company.

- The system provided a more useful final tool. Some tasks that previously required a day of personnel time can now be completed in just one hour.

- The new system is significantly cheaper than its predecessor: the purchase price of the microcomputer hardware is less than one year's data processing bills with the previous system.

It is important to remember the role that the efficiency of the programs play in the usefulness of the overall system: it wasn't the only ingredient in making a useful system, but without it the system was useless.

The background I brought to this project is unusual enough to require an explanation. Although I had programmed for about a dozen years in several languages, I had never programmed in BASIC or on a microprocessor-based system. In the middle of 1981 it finally seemed that the time was ripe to convert the computer system of my parents' business from a card-based mainframe to a personal computer. I viewed building this system as a welcome break from my primary project at the time: I was writing a book entitled *Writing Efficient Programs*. I hoped that building this system would help me refresh my perspective on programming after I had concentrated on the narrow problem of efficiency for so long. Little did I realize that efficiency would prove to be a major issue in this system, and that the material of the book would play a major role in salvaging the system!

The remainder of this paper is divided into four major sections. Section 2 gives an overview of the entire system. After that comes the meat: a detailed study of how the most critical routine was made more efficient is contained in Section 3. The final two sections are a survey of how several other critical routines were improved, and then a retrospective view of the project and some lessons to be learned from it.

1.1. In Defense of Case Studies

This paper describes a case study, and that presents the reader with several problems. The key ideas are intertwined in a large tangle of facts, and while the reader might care about efficiency, he almost certainly does not care about the details of this particular system.

Nevertheless, I think that the reader interested in efficiency can profit by studying this case. The study provides an example of general principles that have been stated in their general form elsewhere (Bentley [1982]); this paper was written because the general form of the principles does not tell the whole story. This perspective is common in other fields: medical journals, for instance, often contain both general discussions of medical conditions and case histories of patients with interesting combinations of maladies. The generalities are important, but the case studies teach practitioners about important facts that do not fit nicely into systematic descriptions. In this paper I discuss how general principles of efficiency were applied in the context of building a particular software system.

Two details of this case study might be of interest to many practitioners of computing. The system provides an example of how a mainframe system can be replaced by a microcomputer system; this activity is becoming increasingly important. Secondly, although efficiency is confronted for a particular system, that system is the widely used Microsoft BASIC that is available on many microcomputers. Although most of the efficiency techniques discussed in this paper are independent of the system on which they are implemented, the few peculiar to this system are broadly applicable to microcomputers.

2. An Overview of the System

The next three sections give a brief overview of the entire system. Although this perspective isn't absolutely essential to appreciate the small pieces of code we will study, it provides the following.

- The context of the system shows the importance of efficiency in some programs and the unimportance of efficiency in others.

- This overview provides an example of how a mainframe system can be profitably replaced by a microcomputer system.

We will consider the system in three stages. We will start by studying the previous system (based on punched cards and an off-site mainframe) and then survey the microcomputer system that replaced it. Finally, we will compare the two systems from the user's point of view.

2.1. The Old System

The company for which I built the system sells the service of designing a questionnaire and then polling several hundred people (typically from 100 to 1200) to record their particular answers on copies of the questionnaire. Once the questionnaires have been filled out, the company faces the data processing task of summarizing the questionnaires in a report that is ultimately delivered to the

customer. The input to that task is illustrated in the partial questionnaire of Figure 1; the output is shown in the report page of Figure 2.

6. A YES vote on propositions 10, 11 and 12 retains the congressional and state legislative district lines as drawn this year. A NO vote requires that the districts be re-drawn. At this time would you vote YES or NO on propositions 10, 11 and 12?

YES -- Retain	1
NO -- Re-draw	2
(Don't know)	3

7. If the Republican primary election were held today, whom would you favor for the U.S. Senate?

Robert K. (Bob) Dornan, U.S. Congressman	1
Barry Goldwater, Jr., U.S. Congressman	2
Pete Wilson, Mayor of San Diego	3
Maureen E. Reagan, Business Executive	4
Paul N. "Pete" McCloskey, U.S. Congressman	5
Other Candidate	6
(Don't know)	7

8. If the Republican primary election were held today, whom would you favor for Governor?

Mike Curb, Lt. Governor	1
George "Duke" Deukmejian, Attorney General	2
Other Candidate	3
(Don't know)	4

Figure 1. A portion of a questionnaire.

The data processing done by the previous system followed a classical approach in survey analysis (for more details on the general problem, see Sonquist and Dunkelberg [1977]); it can be conveniently divided into two phases.

1. **Data Entry and Validation.** The questionnaires were entered into a database that was then checked for consistency and completeness.

2. **Report Preparation.** When the database was complete, it was used as input by a program that prepared the final report.

Table 8.

If the Republican primary were held today, whom would you favor for governor?

	Total	AREA				Recall of Rose Bird	
		North	Cent/ Coast	L.A. County	Other South	Favor	Oppose
Mike Curb	161	47	28	39	47	99	28
Lt. Governor	40.0	43.9	45.2	36.4	37.3	41.1	32.9*
George "Duke" Deukmejian	203	45	31	63	64	121	49
Attorney General	50.5	42.1*	50.0	58.9*	50.8	50.2	57.6*
Other Candidates	3	1			2	1	1
	0.7	0.9			1.6	0.4	1.2
(Don't Know)	36	14	3	6	13	20	7
	8.7	13.1	4.8	4.7	10.3	8.3	8.2
Total Responses	402	107	62	107	126	241	86
	100.0	100.0	100.0	100.0	100.0	100.0	100.0

This table is from a survey of Republican voters taken three days before the June 1982 California primary election. These comments below the tables are usually used for political commentary, but in this example they describe the report format. The first column shows the "raw" response to the question; for instance, 161 of the 402 Republicans polled (or 40.0%) said they favor Mike Curb for governor. The remaining columns cross-tabulate this question with other factors: the next four columns relate to the area in which the voter lives, and the final two columns relate to whether the responent favors or opposes recalling Chief Justice Rose Bird of the California Supreme Court. The percentages in each column sum (to within roundoff error) to 100%; positions with zero voters are left empty. An asterisk next to a percentage shows that the figure deviates from the figure in the total column by more than a fixed percentage (seven percent in this table). In this table, those asterisks draw our attention to the fact that Deukmejian's support is stronger in his home county of Los Angeles than in Northern California, and that opponents of recalling Justice Bird prefer Deukmejian.

Figure 2. A typical report page.

Notice that the two phases communicate only through the common database; besides that, nothing in either phase "knows about" anything in the other.

In the previous system, the questionnaire database was implemented in the classical medium of 80-column punched cards. Each questionnaire was typically represented by a single punched card in which columns 1 through 40 (say) represented the forty single-digit answers to the forty questions on the questionnaire. There were a number of possible variations on this theme, but I will suppress most of those details. For instance, some columns could have contained more than ten answers, and if there were more than eighty columns per record then two or more cards could be used. When such details matter later in this story, I will state the relevant details at the time.

The Report Preparation phase of the system was a fairly straightforward computer program. Given the card database containing the information on the questionnaires and a description of the output format desired, it compiled and printed the information. The first phase, Data Entry and Validation, was more complex. The first part of the phase used a keypunch to record the information on a deck of punched cards. The second part of the phase, Validation, then used a Counter-Sorter to ensure two qualities.

- The information on each card is complete and consistent. Completeness meant that all questions were answered; consistency meant that each response is in a valid range and that certain cross-question conditions are satisfied (for instance, a Republican could not answer a question to be asked only of Democrats).

- The deck of cards comprising the database is complete and consistent. To assist in this task, each questionnaire is assigned a unique identification number before it is seen by the interviewer. Checking that each identification number appears exactly once in the file guaranteed consistency and completeness.

If an error of either type is found on a given card, then the identification number can be used to retrieve the questionnaire corresponding to the card. Identifying the exact tasks performed on the Counter-Sorter in this phase of the existing system proved to be the most difficult task I faced in designing the new system.

2.2. The New System

In this subsection I will briefly sketch the structure of the new system. This overview is only meant to provide enough background to enable the reader to appreciate the relevant details of the programs; in particular, I will not describe the details of the design nor the design space of possible alternatives to this design.

The overall specification of the system remained the same: given the completed questionnaires as input, it must produce a final report as output (using the same format as the previous system). Furthermore, the overall structure of the system was unchanged: the first phase consisted of Data Entry and Validation, followed by a second phase of Report Preparation.

Because the new system was implemented on a TRS-80 Model III microcomputer[2], the questionnaire database was implemented as a disk file. After I explored a number of design

---

[2] A number of factors contributed to my choice of a TRS-80 Model III: it has reliable hardware and software, the customer support is excellent, and the price of a bottom-line machine suitable for data entry was half that of its competitors.

alternatives, I settled on a representation using a "random mode" file in which the first record contained information about the file. I used the unique identification number of each questionnaire as the key of the record representing the questionnaire; questionnaire number *N* was stored as record *N* + 1(questionnaire number *N* was then accessed by a statement like `GET DATABASE,N+1`).[3] A simplified view of the resulting system is shown in Figure 3 (that figure captures the main system flow but ignores the details of half a dozen other programs for various tasks, and several files of minor importance).

The Data Entry and Validation Phase of the system is contained in programs KEYIN and QEDIT. Program KEYIN allows an operator to enter the data in questionnaires onto a cassette tape. That program can run on a 16K Model III with a cassette recorder; no disks are required. Data validation is provided by the program QEDIT (for Questionnaire EDITor); it runs on a 48K, 2-disk Model III. In addition to loading tapes prepared by KEYIN onto the diskette, the program also provides the completeness and consistency checks for both inter- and intra-record verification. If illegal data is found, the program allows it to be changed without rekeying entire records. The program also provides a number of "bookkeeping" functions, such as printing records and initializing files.

The workhorse of the Report Preparation phase is the program PRINT, which runs on a 48K, 2-disk Model III. Its primary input is the Questionnaire database maintained by QEDIT. The other input to the PRINT program is a "description database" that contains a description of the questionnaire format and the print format desired in the report. The description database is prepared using the standard Model III editor SCRIPSIT; it is typically prepared on a 48K, 2-disk system, but it can be prepared on a smaller 16K computer and then subsequently loaded to disk on the larger machine.

The new system makes use of two kinds of Model III computers. The more mundane tasks can be performed on 16K cassette-based systems that sold in mid-1981 for about $1000; that low price means that additional data entry stations can be acquired cheaply when needed. The more complex QEDIT and PRINT programs require a 48K, 2-disk system that sold in mid-1981 for about $4000 (including a production-quality line printer); off-loading data entry tasks onto the small, cheap systems keeps the large system free for running the two large programs.

I purchased a 48K, 2-disk Model III in August 1981, and spent about twenty hours over the next several months learning the computer system. In December 1981 I spent most of a two-week period

---

[3] When I write a BASIC program I will sometimes take the liberty of extending the language to make programs more readable; in particular, I will often ignore the restriction to two-character variable names.

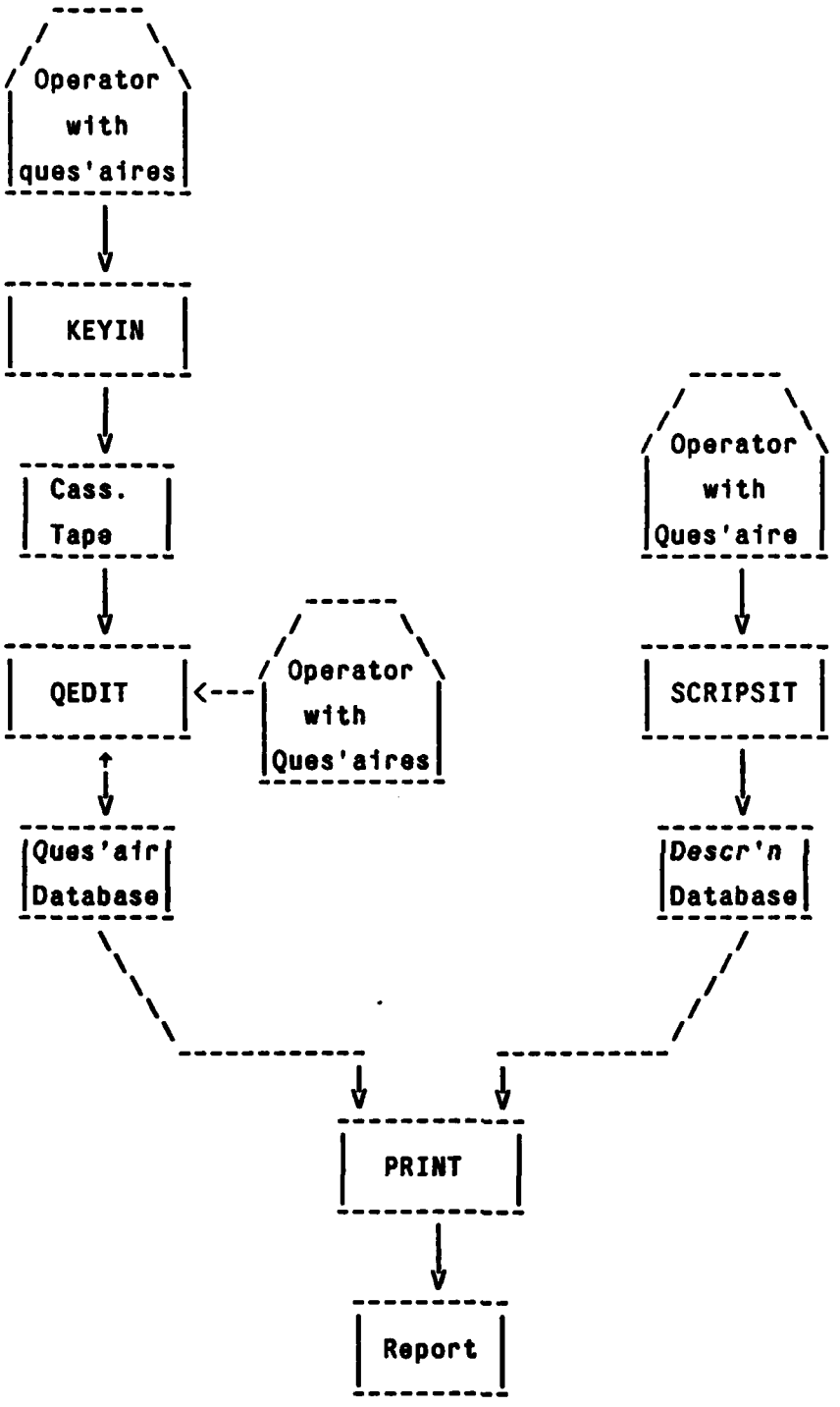


Figure 3. System flow chart for the new system.

implementing the software system I just sketched. That effort resulted in four programs (for a total of about 1500 lines of BASIC code) that formed the nucleus of a correct system with a clean user interface. It suffered from one serious flaw, though: it was too slow. In particular, the PRINT program required fourteen hours to prepare the report for a large study. Over the next month I spent about

forty hours increasing the performance of that program and enhancing the functionality of other programs in the system; over the next several months I spent about eighty hours adding 500 lines of code to the system in enhancements and performance improvements to existing programs and in new programs. (Most of the interesting performance improvements are sketched below.) That work turned the "feasibility study" that was the first system into a useful tool consisting of about a dozen programs in 2000 lines of BASIC code. From January 1982 to April 1982, the company had to use the previous system to process only one poll; by May 1982 the new system had all the capabilities of the previous system. At that time the company was able to retire its array of punched card equipment.

2.3. An Evaluation of the Two Systems

Before concentrating on the technical details of the programs, I would like to take a larger view of the new system as compared to the old. Only the many advantages offered by the new system justify the great deal of work that went into making it more efficient; concentrating a similar effort on an unimportant program would have been a waste of time.

The primary motivation for building the new system was to achieve the following goals.

- **Provide On-Site Computing.** The primary weakness of the existing system was that it made the company dependent on external computing. That had a large cost both in transporting data from the company's office to a mainframe computer installation, and in scheduling time on the mainframe. It was not unheard of for a fifteen-minute computer run to require an employee to drive thirty minutes, wait for an hour or two for a free time slot of fifteen minutes, and then drive thirty minutes back (sometimes only to find that the run contained a single error). The new system removes those obstacles.

- **Increase Security.** The political community is full of tales of campaigns lost because poll results fell into the wrong hands. Even ignoring such drastic possibilities, candidates are rightfully concerned that data for which they pay not be given to their opponents. Processing the data off-site greatly increased the possibility of leaks. For instance, company personnel once watched in dismay as the mainframe operator printed an extra copy of their report for the purpose of adjusting the line printer settings. Having the new system on-site greatly increased the security of the information in the eyes of potential customers.

- **Reduce Computing Costs.** The previous system led to data processing bills of approximately $10,000 per year; the total hardware cost for the new system is less than that amount. The yearly cost for the new system is less than $1000.

The new system met all of those goals. Additionally, the new system has had a number of benefits

that we did not foresee in the preliminary design.

- **Reduce Operator Time.** The user time to prepare the description of the questionnaire dropped substantially. A poll that would take a day to describe on the old system could be described on the new system in an hour (which translates into an *eight-fold* increase in employee productivity). This difference is due both to a more user-hospitable description language (using fundamental tools of computer science) and to the relative ease of using a computer text editor compared to a keypunch. Operator time was also reduced in the data entry task of keying the data on the questionnaires. The new system allows the operator to see what he has typed as he is typing it and to make corrections to the data with little effort; those abilities increase the operator's confidence and therefore his speed.

- **Reduce Processing Time.** The turn-around time to process a survey once the questionnaires were complete dropped substantially. Small "tracking" polls of a hundred questionnaires that previously required one day of processing can now be completed in less than four hours; large studies that previously required four days can now be completed in two days. This is due to the execution speed of the system, the ease of using the KEYIN program compared to a keypunch, and removing the travel time to off-site mainframes. Additionally, the QEDIT program produces cleaner data than that produced by using the Counter-Sorter, which removes the need to make one or two "quick and dirty" runs to find bugs in the input data.

- **Improve Esthetic Appearance of the Report.** The printer on the new system provides output that is easier to read than the old printer. On the old system, the final report was a mixture of computer output and typed summary; in the new system, the entire report is produced on the same computer printer. (This increased another aspect of employee productivity by using word processing tools rather than a typewriter.) Additionally, the new system uses the primitive graphics facilities of the system line printers to present data in graphical form that was previously presented as large tables of numbers; the new form is much more effective in communicating the data.

- **Reduce Size for Computing Equipment and Storage Media.** One of the most expensive items for a small company is office space. Although the new system provides more functionality on-site than the old, it does so using less office space for the machinery. In the old system, each questionnaire database was stored in a container the size of a shoebox, while in the new system it requires just one or two five-and-a-quarter-inch diskettes. The space required to store the surveys done in a year was therefore reduced from a three-foot by four-foot section of wall space to a six-inch by six-inch by one-foot box.

In all these dimensions, the new system is dramatically superior to the old. It is important to remember, though, that all of these advantages would have been for nought had the system been too inefficient to use.

3. Increasing the Performance of PRINT

We will now turn our view from the entire system to just one program -- the PRINT program that generates the final report. Although this program contains just 600 lines of BASIC code (out of a total of 2000 lines in the entire system), it accounted for most of the run time of the system. To study this program, we will first consider its overall structure, and then turn to increasing its time and space efficiency.

3.1. The Structure of the PRINT Program

The purpose of the print program is illustrated in Figure 3: it reads the questionnaire database and description database files, and produces the final report that consists of a series of pages like that in Figure 2. The program is divided into three phases. It first reads the description database file and processes that information into several data structures. The second phase then reads the questionnaire database file once, computes the various figures desired, and stores them in a data structure called the Tally Table. The third phase re-reads the questionnaire database file, and uses that in conjunction with the Tally Table to produce the final report.

The first time the program was used, the first phase required five minutes and the third phase required forty-five minutes, while the second phase required almost fourteen hours. Because the second phase was the "hot spot" of the program, we will concentrate on its structure. Simplified versions of the relevant inputs, outputs, and data structures are shown in a toy example in Figure 4.

The questionnaire shown has only three questions, numbered 1, 2, and 3; the responses to those questions are located in columns 5, 6, and 7 of each record (the first four columns contain the questionnaire number). A description of the questionnaire (similar to the text in the figure, but containing more information) is entered into a description database file using SCRIPSIT. The first phase of the program processes the description database into the Table Definitions of the figure. That data structure says that there are three questions. The first question is represented in Column 5 of the input record, and it is tallied beginning in row 1 of the Tally Table (which will be defined shortly); similarly, the second question is in column 6 and is tallied beginning in row 4. Another part of the description database defines the kind of "cross-tabulation" (abbreviated "cross-tabs") the user desires. In this example, there are four cross-tab columns: the first represents Democrats (that is,

those respondents with a 1 in column 5), the second Republicans (2 in column 5), the third Carter voters in 1980 (1 in column 6), and the fourth Reagan voters (2 in column 6). The facts in the previous English sentence are represented in the program in the Cross Tab Definitions table.

The heart of the second phase of the program is the Tally Table shown in Figure 4: its purpose is to keep a tally of how many times each response has been seen. Although the Tally Table is implemented as a two-dimensional array, it is logically a three-dimensional "ragged" array, consisting of a sequence of two-dimensional arrays (each two-dimensional array has the same number of columns while the number of rows may vary). In this example, the first such two-dimensional array consists of rows 1, 2 and 3, which represent the answers to question 1 (Party Registration). Row 1 corresponds to the first answer (Democrat), and is exactly the data needed to print the first line of numbers in the sample output page in the figure. That row says that there were a total of 3 Democrats; of those, 3 were Democrats and none were Republicans (of course), and 2 were Carter voters while 1 was a Reagan voter. Rows 2 and 3 correspond to Republicans and Others, in a similar fashion. The "Total" line of the sample output can be computed (in the third phase) by summing the values in each column. The Table Definitions table shows that rows 4 through 6 represent the 1980 vote, and rows 7 through 11 represent the Reagan performance rating.

With this background, we can go further into the overall organization of the program. The first phase reads the description database and builds the tables described here, along with several others. The second phase reads the input file and builds the tally table; we will return to its operation shortly. The purpose of the third phase is to print the pages of the report. The operator specifies what pages to print, and that phase then uses a combination of the data structures, the description database, and the tally table to print the desired pages of the report.

The first task of the second phase is to initialize the Tally Table to contain all zeroes. It then reads the entire questionnaire database; for each record, it updates the Tally Table to reflect that record. The following actions are taken for each record as it is read.

- **Read.** Read the record from disk using the BASIC GET statement described earlier.

- **Convert.** Each record is stored on disk in a packed format to conserve disk space; each consecutive pair of digits is represented in one byte. For instance, the pair (5,7) would be represented by the one-byte representation of the integer 157; in general, (I,J) is represented by 100 + 10*I + J in a single byte[4]. This routine unpacks the record into an

[4] The value 100 was added to 10*I + J to ensure that the encoded values did not assume the values of any special control codes when the records were stored on tape in the same format.

QUESTIONNAIRE

1. What is your party registration?
   1.) Democrat
   2.) Republican
   3.) Other
2. For whom did you vote in 1980?
   1.) Carter
   2.) Reagan
   3.) Other
3. How would you rate President Reagan's job performance?
   1.) Excellent
   2.) Good
   3.) Poor
   4.) Very poor
   5.) Don't know

INPUT DATA

```
0001113
0002221
0003125
0004223
0005114
```

DATA STRUCTURES

Cross Tab Definitions

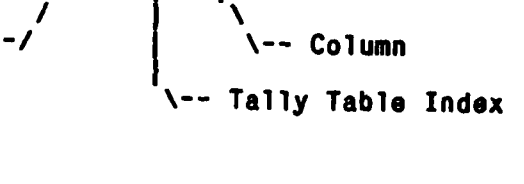
	1	2	3	4	
	5	5	6	6	Column
	1	2	1	2	Value

Tally Table

	0	1	2	3	4
1	3	3	0	2	1
2	2	0	2	0	2
3	0	0	0	0	0
4	2	2	0	2	0
5	3	1	2	0	3
6	0	0	0	0	0
7	1	0	1	0	1
8	0	0	0	0	0
9	2	1	1	1	1
10	1	1	0	1	0
11	1	1	0	0	1

Table Definitions

	1	5
	4	6
	7	7



OUTPUT PAGE

1. Party Registration.

	Total	Registration Dem	Registration Rep	--1980 Vote-- Carter	--1980 Vote-- Reagan
Democrat	3	3	0	2	1
Republican	2	0	2	0	2
Other	0	0	0	0	0
Total	5	3	2	2	3

Figure 4. Components of a sample study.

array of integers.

- **GenerateDouble.** Some questions have more than ten possible answers; when that happens, the user can specify that two columns can be used to hold the answer. For instance, if column 41 of the input contains a 3 and column 42 contains a 2, then referring to the double column starting in column 41 yields the result 32. This routine makes a pass over the array of integers computed by the previous phase, and stores all possible double column values in a separate array.

- **GeneratePseudo.** There are (possibly) additional columns called "pseudocolumns"; this phase generates all pseudocolumns needed. Because this operation does not require much time, I will not describe it in detail.[5]

- **Tally.** The previous operations prepare all the data so it can be readily accessed; this operation uses the prepared data to modify the tally table. Its suboperations are the following.

    - Calculate which cross-tab columns are active for this record. For instance, for the first record in the sample study, columns 0, 1 and 3 are active because this is a Democrat who voted for Carter in 1980. (Column zero is the total column; it is always active.) This data is recorded in the Active Cross Tab array; an active column has the value 1, and an inactive column has the value zero. For instance, the Active Cross Tab array for the first record in the sample study is (1,1,0,1,0).

    - Process each question in the Table Definition array. For a given question, use the column definition to look up its current value in the appropriate array. Then use that value together with the Tally Table index to choose one row of the Tally Table to modify. Add the Active Cross Tab array (as a vector) to that row.

At the end of the above process, the record has been accurately recorded in the Tally Table.

Although this section has covered the second phase of the PRINT program in broad strokes, I have not discussed many of the details (such as how the Double Columns are stored). Some of those details will be given in the discussion to follow.

---

[5] Pseudocolumns allow the user to "generate" information in a record that was not there previously but could be imputed from existing data. For instance, the user might want to identify "high-propensity" voters as respondents who voted in the last two elections and expressed "some" or "a great deal of" interest in the current election. This feature replaced several aspects of the previous system (such as program flags and gang-punching with key punches) and also gives the user additional power without having to write new program features.

3.2. Increasing PRINT's Time Efficiency

As I said earlier, the first version of the PRINT program was *almost* perfect: it produced the correct output in a handsome form, but it took too long to be bearable. I will now describe the series of actions I took to speed up the program.

The first step in improving a program's performance is to *monitor* the program on typical input data to identify the "hot spots" where it is spending its time; we can then improve the performance of those routines. The first problem in using this approach is in selecting "typical" input data. In some systems, such as game-playing programs, very small perturbations in the input can lead to extreme differences in run time. In this program, fortunately, there is not much variation from input record to input record or from survey to survey. The critical parameters are the number of columns per input record and the number of records per survey. The first production survey run under the new system had 1200 records and 80 columns per input record; we will use that particular study as the typical input throughout the rest of this section. This survey is larger than about 90% of the surveys the company typically performs, and the remaining 10% aren't much larger than this. If the program performs well on this input, then it will perform well on most surveys.

When I first ran the program, I was able to use a wall clock to get a rough profile of its run time. This showed that the first phase (reading the description) used just five minutes, while the third phase (printing the report) required forty-five minutes. The five minutes of the first phase was too small to consider improving, while the forty-five minutes for printing couldn't be improved much: that was almost as fast as the printer could go. I had estimated that the second phase would require a few minutes;[6] I was shocked to find that it actually took almost fourteen hours. The part of the second phase devoted to processing the input consisted of 66 lines of code. To improve the performance of that code, I had to monitor it again. This time, I used the clock of the Model III to print out the total number of seconds used for each record after each of the routines mentioned above. This resulted in the following profile of run times.

---

[6] My estimate of the run time of the second phase was based on an IBM System/360 (Model 22) assembly language program that I had written in 1973 to perform a similar function in survey analysis. The corresponding phase in that program completed in about a minute, and the processing speed of the Model 22 is somewhat less than that of the Z-80. My estimate of the run time was off by three orders of magnitude because I had neglected to account for the slowness of the BASIC interpreter.

Read	0.5 seconds
Convert	3.5
GenerateDouble	4.0
GeneratePseudo	0.5
Tally	32.5
Total	41 seconds

This profile shows that each record required 41 seconds to process; processing the entire file of 1200 records required thirteen and two-thirds hours. (Each second of time on an individual record translates into one-third of an hour when applied to the entire set of records.)

Glancing at the above profile shows that the vast majority of the time is spent in Tally; to reduce the time of the phase, we should reduce the time of Tally. Although that was my long-term goal, I started by trying to reduce the times of two other routines: Convert and GenerateDouble. There were two reasons for this detour: I would have to speed up the routines eventually, and these routines were easier to modify (and I needed the practice before I moved on to the hard part!).

I first concentrated on reducing the time spent in GenerateDouble, whose purpose was to compute the values of all possible double columns. The original time of the procedure was 4 seconds; by *fine-tuning the loop that performed* that task, I was able to reduce its time to 3 seconds (which gives a savings of twenty minutes for the whole survey). I then realized that there was a much more promising approach: I was evaluating all 80 double columns, even though only two were needed on this particular survey (and we have never seen a survey that uses more than five double columns). Furthermore, which two were needed could be deduced by investigating just three data structures built during the first phase. I therefore modified the program to have a new routine at the start of the second phase (before any records are read) to scan those structures and make a new structure containing all the needed double columns. As each record was read, the new GenerateDouble procedure would expand only those columns that were needed. This change reduced the time of the procedure from 3.0 seconds to 0.5 seconds; it also reduced the code in the procedure from 7 lines to 6 (although there were now 19 additional lines to build the new table). The net result of this change was to reduce the time by one hour and ten minutes (along with introducing a subtle bug that surfaced only eleven months later; it required several hours to locate and one hour to fix by adding one `if` statement to a line).

This speedup is based on two general rules described in the appendix. The more general is Procedure Rule 2 (Exploit Common Cases); it states that "procedures should be organized to handle all cases correctly and *common cases efficiently*". The modified GenerateDouble routine is in fact

slower than the original if there are many double columns. but is much faster on common cases. The specific rule that told us how to achieve that speed is Space-For-Time Rule 4 (Lazy Evaluation), which states that "the strategy of never evaluating an item until it is needed avoids evaluations of unnecessary items". In this case we avoid the cost of evaluating unnecessary double column values. A list of efficiency rules can be found in the appendix; throughout the rest of the body of this paper, the rules will be referred to when they are used by a reference of the following form: *(Principles: Space-For-Time Rule 4 (Lazy Evaluation); Procedure Rule 2 (Exploit Common Cases).)*

I next concentrated on the procedure Convert; its main loop unpacked a number N = 100 + 10*I + J to the two numbers I and J (both between 0 and 9). The obvious solution to the problem would subtract 100 from N, and then use division to compute I, and finally use multiplication and subtraction to compute J. In the preliminary design of the program I realized that having an input loop do a divide, a multiply, and several other operations on each character would have been far too expensive. For that reason, and for ease of coding, I chose instead to use a two-dimensional array called the Translate Table that facilitated the translation. In this table, for instance, the following relations hold.

```
TranslateTable(54,0) = 5
TranslateTable(54,1) = 4
```

Thus the two desired digits could be found in `TranslateTable(N-100,0)` and `TranslateTable(N-100,1)`, without using any multiplications or divisions. This was in the first version of the program; the time to translate 40 bytes (80 columns of data) was 3.5 seconds. My measurements showed that indexing two-dimensional arrays was almost twice as expensive as indexing one-dimensional arrays, so I changed the structure to two one-dimensional tables; this removed two subscript operations in the inner loop, and reduced the time of the routine to 2.5 seconds. *(Principles: Space-For-Time Rule 2 (Store Precomputed Results); Expression Rule 4 (Pairing Computation).)*

Those little changes saved a total of 4.5 seconds per record, or an hour and a half of the fourteen. It was now time to attack the real hot spot: Tally. The first task of Tally is to compute the Active Cross Tab array described earlier; because that is extremely fast, I will not describe it. The second task performs the actual tallying; in high-level pseudo-BASIC, it looks something like this:

```
for I = 1 to QuestionCount
    Row = TallyTableIndex(I)-1+Value(Column(I))
    for J = 1 to CrossTabCount
        add ActiveCrossTab(J) to TallyTable(Row,J)
    next J
next I
```

The action of the above code is simple: for each question, it computes the appropriate row in the Tally Table, and then adds the Active Cross Tab array (each element of which is either a zero or a one) to

that row.

Although this code is clear, it performs many needless actions in adding zeros to Tally Table elements. In the sample study, it was typical to see that of 11 possible cross tabs, only four were active (and these values were typical of most studies); therefore, eleven additions were performed when four would have sufficed. There are several ways in which we can avoid that unneeded work; the following code shows one of the simplest approaches.

```
for I = 1 to QuestionCount
    ActiveRow(I) = TallyTableIndex(I)-1+Value(Column(I))
next I
for J = 1 to CrossTabCount
    if ActiveCrossTab(J) = 1 then
        for I = 1 to QuestionCount
            add 1 to TallyTable(ActiveRow(I),J)
        next I
next J
```

This code is only a little more complex than the previous code and uses only a little more storage. It first stores in the Active Row array all rows to be modified. The next pair of `for` loops are inverted from their previous order: the outer loop examines the columns, and only if a given column is active does the program then add one to the appropriate rows (which can be found in the Active Row array).

This code reduces the run time in two ways: it makes only four additions rather than eleven, and those four do not involve accessing the Active Cross Tab array. The result of eliminating these operations (and the interpretation of the source code containing the operations) is to reduce the time of Tally from 32.5 seconds to 14 seconds, and the time for the overall survey from 12.3 hours to 6 hours. The change increased the number of lines of code from 9 to 15. This change also adds a new suboperation to Tally: in addition to computing the Active Cross Tab array and performing the tallying, it must now also compute the Active Row array. Monitoring showed that of the 14 seconds spent in Tally, 5 seconds are spent in preparing the Active Row array and 9 seconds are spent in performing the additions. *(Principles: Logic Rule 3 (Reordering Tests); Expression Rule 2 (Exploit Algebraic Identities).)*

At this point, each record required 18 seconds of processing, and half of that was devoted to performing the additions in Tally. The nine seconds translate to three hours of time for the entire survey, so it is well worth concentrating on the following lines.

```
for I = 1 to QuestionCount
    add 1 to TallyTable(ActiveRow(I),J)
next I
```

After trying several ways to increase the speed of the above loop in BASIC, I finally broke down and

rewrote it into Z-80 assembly code. The assembly code consisted of 29 instructions: after translating and testing it using small driver programs, I stored it in the PRINT program as an array of 22 two-byte integers. The three lines of code in the above Tally loop were replaced by five lines to call the subroutine plus three additional lines to store the assembly code in the BASIC program. This change decreased the time for the loop from nine seconds per record to one second per record. This resulted in the following profile of run times.

Read	0.5 seconds
Convert	2.5
GenerateDouble	0.5
GeneratePseudo	0.5
Tally	
Prepare Active Row Array	5.0
Perform additions	1.0
Total	10 seconds

The ten seconds per record translate to a total of three hours and twenty minutes for the entire survey. *(Principle: Work at the lower design level of assembly coding.)*

The hot spot for the second phase is now the part of Tally that prepares the Active Row array. I did a number of experiments with a small program that performs that task, and I was confounded to see that the experiments predicted that the code would require three seconds, while my measurements showed that it took five seconds. After looking at a number of possible explanations, I finally found the problem: there were only a few variables in the small test programs, while the complete PRINT program had about one hundred variables. Furthermore, the programs in the hot loops didn't use their variables until well into execution of the program, so they appeared near the end of the BASIC interpreter's search list of variable names. I fixed this problem by adding one new line to the start of the program that assigned a zero to each of those critical variables; that caused the variables to appear at the front of the search list. This change sped up various parts of the program, which resulted in the following profile. *(Principle: Work at the lower design level of system-dependent techniques.)*

	Original	Final
Read	0.5 seconds	0.5 seconds
Convert	3.5	1.5
GenerateDouble	4.0	0.5
GeneratePseudo	0.5	0.5
Tally	32.5	
Prepare Active Row Array		3.0
Perform additions		1.0
Total	41 seconds	7 seconds

While the original program required thirteen hours and forty minutes to build the Tally Tables, the final version of the program can do the same job in just two hours and twenty minutes.

There are a number of possible alternatives that might speed up the program at this point. The two main hot spots are now Convert and Prepare Active Row array; the times of those procedures could be reduced either by working in BASIC or by converting the routines to assembly code. I chose to take a different approach to the problem. Observing the typical use of the program showed that for any particular survey, its Tally Table was computed three times: the first to get a rough draft, the second to make comments on bottom of the output pages (using a feature in the third phase of PRINT that I haven't described) and the third to make extra copies. For a survey of this size, three runs would spend a total of seven hours computing one Tally Table. I therefore included commands that allow the operator to save Tally Tables to disk and to load them back from disk to avoid their recomputation. This required about sixty lines of straightforward code, and ensured that each Tally Table was computed just once. *(Principle: Expression Rule 1 (Compile-Time Initialization).)*

This is where this part of the story ends. In its first version, the second phase of the program was executed three times for a total of forty-one hours; that was too much time to be useful. In the final version, the phase was executed once at a cost of a little over two hours. Although that is still a significant time to wait, it was sufficient for the company. I am sure that by spending a great deal of my time, I could have reduced that running time even further, but this was "good enough". The cost of this speedup was not great: reducing the time of the program by a factor of almost six required replacing 66 lines of BASIC code with 112 lines of BASIC and 29 lines of Z-80 assembly code (and many of the BASIC lines were unchanged from the original 66), and adding the ability to store and retrieve Tally Tables required sixty lines of code. The changes required a total of about forty hours of my time over a period of several weeks.

3.3. Increasing PRINT's Space Efficiency

You always notice how much time a program takes; you can tell by the wait. Space is different: if the program uses less space than what is available on the machine, everything is fine. When it uses too much space, though, you find out immediately: the program won't run. Although that didn't happen to PRINT on the first survey it processed, it did happen on a later survey: there wasn't enough space on the 48K Model III to store the Tally Tables. When this occurred, I made two modifications to reduce the space of the program.

The first change i made was to replace all the error messages in the program (about forty) by error numbers. I then used SCRIPSIT to prepare a list of the message numbers and the corresponding text. This saved about 850 bytes of program memory by replacing a string of twenty characters (on the average) with a decimal integer. This had the unexpected advantages of allowing me to write somewhat more informative error messages, and also gave the operators a convenient place in which they could make notes about the messages. *(Principle: Time-For-Space Rule 1 (Packing).)*

The next change led to a much greater savings. My original program text was very nicely structured: it included numerous comments and many blank spaces to reflect the program structure. This made the program easy to write, debug, and maintain, but consumed a fair amount of storage at runtime. I therefore used the Disk BASIC CMD "C" (for "compress") to remove all the comments and unnecessary spaces from the program; this reduced the program's space requirements by 4300 bytes (and left its running time unchanged). Although I desperately needed that space, I was not about to sacrifice the clarity of comments and blank spaces. I therefore maintain two copies of the program: the clean version is the master copy, and after making a change in it I use the compress command to get the working copy. This slightly increases the difficulty of modifying the program (I have to apply the compress command and store the resulting program), but it combines the best of both worlds: the program is both easy to maintain and space-efficient.

These two changes together save about 5100 bytes of storage, or 2550 16-bit integers. Because Tally Tables are allocated to have 11 column (one total column and ten for cross-tabs), this gave the space for 230 additional rows of the table. The program originally ran out of space when 350 rows were allocated to the Tally Table, so I added an additional 150 rows (for a total of 500) -- that has been large enough for all surveys we have seen. The remaining extra space (almost 2000 bytes) was used for other program enhancements.

4. Increasing the Performance of Other Programs

In the previous section we examined the performance of the PRINT program in some detail. In this section we will survey at a more superficial level the performance issues that arose in several other programs.

4.1. KEYIN Performance

The purpose of KEYIN is to allow the operator to enter the data from questionnaires onto a cassette tape; KEYIN typically runs on a 16K Model III with a cassette recorder (no disks are required). In its initialization phase, KEYIN asks the operator how many columns there are per record in this survey. After that, KEYIN behaves like a very simple screen editor for records that consist of that many decimal digits. When the operator types a digit to the editor, that digit appears on the screen and the process continues; this is the typical operation. The operator can also perform such operations as backspacing over previously typed digits, retyping digits, and forward spacing over digits. When the operator has typed all the digits in the record, hitting the ENTER key will store the record in main memory and cause the operator to be working on a new (initially empty) record. If the operator types ENTER in the first column of the new record, that allows him to dump all the records currently held in memory to a cassette tape.

The first performance problem I faced in KEYIN was of a different character than the problems we saw in PRINT. In PRINT, the problem was that a single operation took more than half a day to complete. In KEYIN, the problem was that an operation took less than half a second to complete, but that was still too long. The operation was getting a new character from the keyboard (using the BASIC `INKEY$` function), and it took so long that if the operator typed several characters quickly, then the program would "drop" some of them -- they wouldn't appear on the screen. This is a disaster for a data input program: that behavior destroys the trust the operator should have in the program, and causes him to type at a speed far below his capability. Inspection of the code quickly showed that the problem was in the part of the code that mapped a buffer position into a screen position. The problem arose because the characters were stored internally in an array of 250 characters, but were displayed on the screen in five rows of fifty characters each. The characters on the screen were addressed by numbers from 0 (for the character in the upper left corner) to 1023 (bottom right), with each row containing 64 characters. The character in buffer position $N = 50*I + J$ (where N is between 1 and 250 and J is between 1 and 50) should be displayed on the screen in character position $200 + 64*I + J$. That is the process that was consuming the time: given a number $N = 50*I + J$ (where I is between zero and four), compute the number $200 + 64*I + J$.

The first way I solved the problem was with a series of a division, multiplications, and additions. Although that method was straightforward, it was too expensive in time for two reasons: the division and multiplication operations are very time consuming, and there is a great deal of program text to be interpreted in performing the operations. I therefore replaced those operations with a table that was initialized by the following code.

```
for I=0 to 4
    for J=1 to 50
        ScreenPosition(50*I+J) = 200 + 64*I + J
    next J
next I
```

After this, the screen position corresponding to buffer position N (where N is between 1 and 250) could be found in `ScreenPosition(N)`. The resulting program was fast enough to keep up with the typing of the fastest operator. *(Principle: Space-For-Time Rule 2 (Store Precomputed Results).)*

The next performance problem in KEYIN showed up when the operator typed ENTER at the end of each record. The task of the program at this point was to pack the record: the 250 digits of the record were packed into a 125-byte string (the pair of digits (I,J) was represented by 100 + 10*I + J -- this packing was used throughout the system). The initial version of the program always packed 250 digits, even though far fewer were typically used (even large surveys had only 80 digits per record). The time required to pack the first record was five seconds; packing the last record required forty-five seconds (the increase was due to the added overhead of the BASIC interpreter reorganizing the strings in memory). I therefore changed two lines of code to pack only the digits actually used, rather than packing all 250 digits; this reduced the time to process each record to less than a second. In the first version of the program the five-second wait between records was irritating and the forty-five second wait was immobilizing (it cut operator productivity in half); the wait in the modified program was not noticed by the operator. *(Principles: Space-For-Time Rule 4 (Lazy Evaluation); Procedure Rule 2 (Exploit Common Cases).)*

This change had two further advantages. First, the smaller records required less space in the 16K memory, and more records could therefore be stored before dumping them onto a cassette (over 100 instead of 50). Second, shorter records were written to the cassette tape; this sped up the time-consuming (and error prone) operations of reading and writing tape by a factor of almost two.

The first implementation of KEYIN suffered from a serious human interface problem: if the operator missed a digit, he would not discover this fact until he typed ENTER, only to find that he wasn't at the end of the record. In the original system this problem was solved by setting "skip columns" at the columns corresponding to the end of each of the (typically five) pages of the questionnaire; when the

keypunch operator turned the page he would also hear the keypunch skip a column. Thus if an operator ever got "out of synch" with the questionnaire, he would discover that fact at the end of the current page. The lack of a corresponding feature was a serious drawback to the new system, so I decided to add a corresponding feature to KEYIN called "whistle columns". At the initialization of the program, the operator gives the program a set of whistle columns, and whenever the program enters one of those columns as the operator is entering data, a short whistle is sounded through the mini-amplifier.

During the design of this feature it became clear that the main performance problem was going to be deciding whether the current column is a whistle column; that operation is performed once for every keystroke. I first considered solving the problem by searching an array of (typically five) values of the columns, but that was far too slow. I investigated a number of alternatives, such as performing a binary search in the array or caching the next whistle column, but all were either too slow or too complicated. I finally chose to solve the problem by a 250-element array called Whistle Column in which each entry is true only if that column is a whistle column. That gave a program that was fast enough to keep up with all operators, but it used 500 more bytes of space (of the 16K byte machine). *(Principle: Logic Rule 4 (Precompute Logical Functions).)*

I solved the space problem by packing both the Whistle Column array and the Screen Position array into a single array of 250 integers, which I will call the Packed Array. The values of that array were initially the same as the Screen Position array described earlier; the values in the whistle columns were negated during initialization. Thus the values of the two logical arrays could be found in the one actual array by the following conventions.

```
ScreenPosition(I) = abs(PackedArray(I))
WhistleColumn(I)  = PackedArray(I)<0
```

This program was fast enough to keep up with all operators, and the packing saved 500 bytes of storage (or enough space for a dozen more records). The modification to include whistle columns required six new lines of code and changes to three existing lines of code. *(Principle: Time-For-Space Rule 1 (Packing).)*

4.2. QEDIT Performance

The first version of the QEDIT program offered the operator a selection of about ten commands; all of those but one were able to process a file of 1200 records in less than ten minutes (and most were used on just a few records at a time, and almost always on fewer than a hundred records). One operation stood out as a performance problem: the "Verify" command that performs intra-record consistency checking. That operation performs a number of tasks: it ensures that all digits are

present, and that each digit is in an appropriate range. Furthermore, it checks properties such as that Republican-only questions cannot be asked of Democrats, and that if a voter names Smith as his favorite candidate in a field, then he cannot name him as his second-favorite or least-favorite candidate. (These conditions are encoded in a simple language understood by QEDIT; the language includes the "pseudocolumns" used by PRINT.)

This verification operation required fifteen hours on the first survey on which it was used (the same survey described in the previous section). Unlike in my experience with PRINT, though, this survey was highly atypical: it requires twice as much time as any survey we have seen since. Even so, the program was still unreasonably slow. I made two changes to reduce its running time.

- I learned from my experience in the PRINT program that initializing frequently used variables to zero at the start of the program could increase the program's performance. Doing this to the hot variables in the Verify operation decreased its run time from fifteen hours to seven hours.

- Most of the run time of the program was spent in sixteen lines of straightforward loops and logic to evaluate "pseudocolumns". I spent several hours fine-tuning those sixteen lines to remove unneeded iterations (by determining when a value could no longer change) and boolean variables (by exiting a loop in a certain way). Those changes reduced the code to just eleven (incredibly subtle) lines, and reduced the run time from seven hours to five hours. *(Principles: Logic Rule 2 (Short-circuiting monotone functions); Logic Rule 5 (Boolean variable elimination).)*

The five hours of the final code was "fast enough" for two reasons. First, even on most very large studies, the total time required for the operation was less than two hours. Second, that time did not have to be spent in one block; rather, each batch of records could be verified as they were read from tape.

The other performance problem in QEDIT arose in the "Figure" command. That command was added to the orginal design so the operator can figure preliminary results that can be given to clients in tight races before the final run of the PRINT program. It was essential that this operation be very efficient. During the first design, calculations showed that this operation would take at least an hour to run on large surveys. Most of that time was spent in unpacking the entire record, even though only a few of the columns were used. I therefore used the same technique I used in PRINT's GenerateDouble routine of only expanding the needed columns. That reduced the time of the operation to ten minutes. *(Principles: Space-For-Time Rule 4 (Lazy Evaluation); Procedure Rule 2 (Exploit Common Cases).)*

4.3. Report Programs Performance

We will now briefly consider the performance problems that arose in three programs not shown in Figure 3; all programs dealt with various parts of the final report that was delivered to the client. The performance problems we will study are much less serious than those we have seen previously; the problems in PRINT and KEYIN threatened the viability of the system, and the bottleneck in QEDIT was a serious problem in its use. On the other hand, the problems we will now see were not critical: the changes mitigate some minor irritations, but they weren't essential to the usefulness of the system.

The PRINT program allows the user to provide "annotation files" so that explanatory comments may be printed under the tables. In the original card-based system these comments were prepared on a typewriter and then laboriously justified by hand and transferred to punched cards. In the first version of the system, the operator was happy to prepare the file using the SCRIPSIT text editor, doing all the justification by hand; that was much faster than using punched cards. I soon realized that this was a misuse of employee time, so I wrote the program JUSTIFY to justify text in annotation files. The original version took fifteen minutes to process a large file. Monitoring showed that more than half the time of the program was spent in a one-line loop that scanned the current input string looking for the end of the current word. I therefore modified the program to use the BASIC `INSTR` (for "in string") function to locate the next blank in the input buffer (and thereby the end of the current word); that reduced the run time to eight minutes. Several additional small changes reduced the time to six minutes. *(Principle: Loop Rule 2 (Combining tests).)*

The purpose of the PRINTGRF program is to use the simple graphics capabilities of the system line printers to produce graphs that summarize the data in the reports. The original implementation required eight minutes to print a one-page graph on a Radio Shack Model V Line Printer. Almost all of the program's time was spent in an eight-line loop; modifying the loop to notice a common case and process it in a straightforward way reduced the program's run time to three and a half minutes. That time cannot be improved, because the line printer takes that long to print a page using half-spacing (in which there are 132 lines per page). *(Principle: Procedure Rule 2 (Exploit Common Cases).)*

The final time-consuming report program is called TOSS; its purpose is to run a simulation that is included in the final report. If the survey consists of eight hundred records, then the simulation involves one hundred experiments in which a coin is "tossed" eight hundred times; the number of heads in each experiment is stored and then graphed in simple form to give the reader an idea of the survey sample error (that graph gives much more of an intuitive feel for the error than simply listing the confidence limits that can be found in any statistics book). In this example, the simulation involves calling the random number generator eighty thousand times (and performing other

bookkeeping operations), which requires approximately an hour. Because that would have been too much time to pay for each survey, I spent a weekend of computer time running simulations for all the common survey sizes (all multiples of one hundred from 100 to 1200, and some multiples of fifty), and stored the results on a single diskette. This allowed the operator to retrieve the results of the simulation in just a few seconds, and the computer didn't have any plans for that weekend anyway. *(Principle: Space-For-Time Rule 2 (Store Precomputed Results).)*

5. Reflections

Those are the facts of the story. They add up to the following: a potentially useful system was made useful by increasing the performance of several programs without seriously detracting from their functionality or maintainability. I'd now like to go beyond the hard facts to reflect on the methodology I used in the system. In Section 3.3 of *Writing Efficient Programs* I described five steps that I claimed were "essential parts of a methodology of building efficient computing systems." In this section I will list those five steps (in bold face), and comment on how they relate to my experience in building this system.[7]

1. The most important properties of a large system are a clean design and implementation, useful documentation, and maintainable modularity. The first steps in the programming process should therefore be the design of a solid system and the clean implementation of that design.

When I designed the overall system, I knew that efficiency would be an issue. I was therefore careful to choose appropriate data structures between programs (such as the questionnaire database) and within programs (such as the Tally Table in PRINT). Most of my effort, though, went into designing a system that could be implemented quickly and correctly and be used by a staff unfamiliar with computers.

When I first implemented the system as a set of computer programs, I was very careful to use absolutely no time-saving tricks: the original code was very clean. As we have seen, some of that code was later made much faster; some readers might think that I should have used more trickiness in the first implementation. I can't begin to describe how glad I am that I didn't. If I had taken that approach, I am sure that I could not have implemented the first system in just two weeks, and the resulting system (if it ever worked) would have been much harder to maintain.

---

[7] The material in bold face is from *Writing Efficient Programs* by Jon Louis Bentley, (c) 1982 by Prentice-Hall, and is reprinted with the permission of Prentice-Hall.

2. If the overall system performance is not satisfactory, then the programmer should monitor the program to identify where the scarce resources are being consumed. This usually reveals that most of the time is used by a few percent of the code.

It has long been a piece of programming folklore that most of the run time of a program is spent in a small part of the code. This occurred several places in this system.

- *The overall system.* The complete system consisted of 2000 lines of code, yet most of the run time was spent in the 600-line PRINT program.

- *The PRINT program.* The total run time of the PRINT program was 14.5 hours; of that time, 13.6 hours were spent in the second phase, and of that time, 11 hours were spent in the Tally routine. Thus in this program, 66 lines of code (a little over 10%) accounted for 94% of the run time, and 3 lines of code (just 0.5%) accounted for 75% of the run time.

- *Other programs.* In QEDIT, most of the time was consumed by the Verify operation (30 lines out of 600). In KEYIN, the problem of dropping characters could be traced to three lines of code (out of 150), and the forty-five seconds to pack a record was due to two lines of code. Eight-line loops in the 60-line PRINTGRF program and the 120-line JUSTIFY program accounted for at least 80% of their run times.

This property of programs allows us to ignore tricks in the first implementation of a system: we shouldn't be clever with most code, because most code just plain doesn't use much run time. For instance, any cleverness I might have expended on speeding up the 540 lines of PRINT not in the second phase would have been wasted: even the simple version didn't use much time.

I used several different techniques to monitor the programs. A wall clock sufficed to time the various phases in most programs. After that, I used the Model III timer to time the individual routines in the expensive phases. In some programs I "sampled" the program execution by hitting the BREAK key and observing the current line number; although this does not give absolutely accurate results, it can help a programmer to get a rough idea of where the time is being spent.

3. Proper data structure selection and algorithm design are often the key to large reductions in the running time of the expensive parts of the program. The programmer should therefore try to revise the data structures and algorithms in the critical modules of the system.

Changing data structures and algorithms can lead to substantial savings. Unfortunately,

I had relatively few chances to make changes at this level in this system. I was able to change data structures in the GenerateDouble routine of PRINT, in the Figure command in QEDIT, and in the Whistle Column array of KEYIN; those modifications all gave substantial speedups. The other parts of the system were all "optimal" in the sense used in most algorithms texts (I spent a great deal of effort in the design of the program to choose structures that would be easy to implement; they turned out to be efficient also), so that field offered little help in this system.

4. If the performance of the critical parts is still unsatisfactory, then use the techniques of Chapters 4 and 5 [of *Writing Efficient Programs*] to recode them. The original code should usually be left in the program as documentation.

The program speedups that I have described in this paper might appear at first glance to be just one trick after another. I think that deeper investigation can show that this is not the case; in fact, they can all be traced to the simple set of rules cited above. To show how the rules relate to the speedups, the Appendix of this paper contains a list of rules from the chapters mentioned above together with a brief description of how each rule was applied in the system. The principles in the text give pointers to the relevant rules.

I did not follow the advice of leaving the original code in the modified program because that would have required too much space in BASIC. Rather, I kept copies of the original code in the documentation folder of each program, together with a brief description of why and how the code had been changed.

5. If additional speed is still needed, then the programmer should work at lower design levels, including hand-written assembly code, operating system modifications, microcode, and special-purpose hardware design.

Only once in this system did I have to resort to the lower design level of hand-writing assembly code. In that case, replacing three lines of BASIC with 29 lines of assembly code reduced the time to process a survey from six hours to three hours and twenty minutes. Before I translated the code, I was very careful to isolate the hot spot into just three lines of BASIC: it would have been much more difficult to translate larger pieces of BASIC code into assembly code.

On several occasions I worked at a design level just one step up from assembly code: I used knowledge about the operation of the BASIC interpreter. Many books and articles on efficient BASIC advise programmers to use such techniques as removing spaces and comments from programs and placing many statements on a single line; simple experiments that I conducted showed that such practice had little impact on the overall program efficiency (and trying such practices on larger programs confirmed this). I did

find one such technique extremely useful, though: referencing frequently used variables at the beginning of the program reduced the time of the second phase of PRINT from three hours and twenty minutes to two hours and twenty minutes, and doubled the speed of the Verify operation in QEDIT.

A design level at which I tried to make changes was the level of translation to machine code. I had hoped that I could replace the BASIC interpreter I was using with a BASIC compiler. Unfortunately, the Radio Shack BASIC compiler supports a fundamentally different language from that supported by the interpreter, and I estimated that converting the system would have required at least two weeks, which was more time than I was willing to spend for the kind of speedups I expected to see. (Although if I had to build this particular system over from scratch, I would seriously considering using a compiled language.)

Acknowledgements

I would like to thank Lawrence Butcher, Tom Lane, Jim Saxe and Guy Steele for their helpful comments on this paper, and the employees of Western View-Point Research, Incorporated, for their help in developing the system.

References

Bentley, J. L. [1982]. *Writing Efficient Programs*, Prentice-Hall, Englewood Cliffs, NJ.

Sonquist, J. A. and W. C. Dunkelberg [1977]. *Survey and Opinion Research: Procedures for Processing and Analysis*, Prentice-Hall, Englewood Cliffs, NJ.

I. A List of Efficiency Rules

This appendix lists (verbatim) seventeen rules for increasing program performance from the twenty-six rules in Chapters 4 and 5 of *Writing Efficient Programs*.[8] The comments after each rule describe their application in this system; most of the applications are described more fully in the text.

Space-For-Time Rule 2—Store Precomputed Results: The cost of recomputing an expensive function can be reduced by computing the function only once and storing the results. Subsequent requests for the function are then handled by table lookup rather than by computing the function.

---

[8] The rules are from *Writing Efficient Programs* by Jon Louis Bentley, (c) 1982 by Prentice-Hall, and are reprinted with the permission of Prentice-Hall.

- PRINT. The Convert routine used two one-hundred element tables to unpack a one-byte representation of two decimal digits.

- KEYIN. The Screen Position table replaced an expensive division, multiplication, and additions (and interpretation of the BASIC code expressing those operations) with a single array reference. This reduction in time allowed the program to keep up with the fastest typist (it had previously dropped characters).

- TOSS. This program took an hour to run a simulation to prepare a table. I therefore spent a weekend of computer time to calculate all tables of common sizes and stored them on a diskette; a stored table could then be retrieved in a few seconds.

Space-For-Time Rule 3—Caching: Data that is accessed most often should be the cheapest to access.

- KEYIN. The prototype design of KEYIN cached the screen position corresponding to the most recently used buffer position; this increased the speed of that procedure, but not enough to keep up with fast typing. Caching the next active whistle column would have been fast enough to solve that problem, but the technique involved very subtle code. In both cases I eventually solved the problem by precomputing all possible answers.

Space-For-Time Rule 4—Lazy Evaluation: The strategy of never evaluating an item until it is needed avoids evaluations of unnecessary items.

- PRINT. The GenerateDouble routine originally generated all possible double columns; modifying the routine to generate only the double columns actually needed reduced its time from 4.0 seconds to 0.5 seconds per record (this gave a savings of over one hour for the complete file).

- QEDIT. The Figure command only expanded the columns it needed; this reduced its run time from over an hour to ten minutes.

- KEYIN. Modifying the program to pack only the digits actually used rather than all possible digits reduced the time to pack the last record from forty-five seconds to less than one second.

Time-For-Space Rule 1—Packing: Dense storage representations can decrease storage costs by increasing the time required to store and retrieve data.

- A common packing to store two decimal digits in a single byte was used throughout the system. In KEYIN this more than doubled the number of records that could be stored before dumping to tape. It almost halved the amount of time required to read and write

the cassette tapes between KEYIN and QEDIT, which also reduced the number of errors. The representation allowed 1300 records to be stored on a diskette, rather than 650. That difference is extremely significant: roughly half the surveys done by the company have more than 650 records (and therefore require one diskette rather than two because of the packing), while only a few surveys in the history of the company have consisted of more than 1300 surveys (and would require more than one diskette).

- KEYIN. The Screen Position array was packed in the same array as the Whistle Column array; this saved 500 bytes on the 16K byte machine.

Time-For-Space Rule 2—Interpreters: The space required to represent a program can often be decreased by the use of interpreters in which common sequences of operations are represented compactly.

- PRINT. During the design of the PRINT program I viewed the entire program as an interpreter. The first phase of the program reads the description database file and translates that into the language of the interpreter (its data structures); the second and third phases then interpret that language.

Loop Rule 2—Combining Tests: An efficient inner loop should contain as few tests as possible, and preferably only one. The programmer should therefore try to simulate some of the exit conditions of the loop by other exit conditions.

- JUSTIFY. The inner loop in the program originally made two tests: "have I come to the end of the input buffer?" and "is this character a blank". I reduced those tests to one by appending an extra *sentinel* word on the end of the buffer, so the inner loop knew that there was always a trailing blank and didn't have to test whether the buffer was exhausted. The test for an exhausted buffer was moved to an outer loop by asking whether the current word is the sentinel word.

Loop Rule 6—Loop Fusion: If two nearby loops operate on the same set of elements, then combine their operational parts and use only one set of loop control operations.

- QEDIT. The routine that performs the functions corresponding (roughly) to Convert and GenerateDouble gained about one second per record by using a single loop for the two operations. Not only did this reduce the loop control overhead, but certain variables could be shared in the loop (rather than being evaluated twice).

Logic Rule 2—Short-circuiting Monotone Functions: If we wish to test whether some monotone nondecreasing function of several variables is over a certain threshold, then we need not evaluate any of the variables once the threshold has been reached.

- QEDIT. The Verify operation was reduced from seven hours to five hours by exiting a

loop as soon as the value of an "OR-ed" variable became true or when an "AND-ed" variable became false.

Logic Rule 3—Reordering Tests: Logical tests should be arranged such that inexpensive and often successful tests precede expensive and rarely successful tests.

- PRINT. This was the idea that led to inverting the loops in the Tally routine: the common case that a Cross Tab column was active was tested earlier in the inverted loop. This reduced the time to process a survey from 12.3 hours to 6 hours.

Logic Rule 4—Precompute Logical Functions: A logical function over a small finite domain can be replaced by a lookup in a table that represents the domain.

- KEYIN. As each digit is input, the KEYIN program must determine whether the given column is a "whistle column", and, if so, give audio output on a mini-amplifier. An implementation that searched even a short array of such columns was too slow; the final program therefore used an array of 250 values that were either true or false (that array was shared with the ScreenPosition array).

Logic Rule 5—Boolean Variable Elimination: We can remove boolean variables from a program by replacing the assignment to a boolean variable V by an `if-then-else` statement in which one branch represents the case that V is true and the other represents the case that V is false. (This generalizes to `case` statements and other logical control structures.)

- QEDIT. The time of the Verify operation was reduced from seven hours to five hours by replacing repeated tests of boolean variables with an early exit from the loop to a place at which the values were known.

Procedure Rule 1—Collapsing Procedure Hierarchies: The run times of the elements of a set of procedures that (nonrecursively) call themselves can often be reduced by rewriting procedures in line and binding the passed variables.

- QEDIT. The original version of the Verify command used a clean hierarchy of routines to access and to expand records (that hierarchy was used by all other routines in the program). The time for reading the records was reduced by having a special routine that performed only the operations needed by QEDIT, without using a series of `GOSUB` commands.

Procedure Rule 2—Exploit Common Cases: Procedures should be organized to handle all cases correctly and common cases efficiently.

- PRINT. The GenerateDouble routine was changed to expand only the double columns actually needed; it is always correct. The new version is slower than the old version if

there are many double columns, but is much faster for typical inputs (the typical reduction was from 4.0 to 0.5 seconds). The same comments apply to inverting the loop in the Tally routine: it might be slower in a few cases, but it is much faster on the average (the typical reduction was from 32.5 to 14 seconds).

- KEYIN. The program used variable-length records to exploit the common case that the records contained far fewer than 250 digits; this more than doubled the number of records that could be stored.

- PRINTGRF. The time to print a graph was reduced from eight minutes to three and a half minutes by efficiently handling a common configuration of input characters.

Expression Rule 1—Compile-Time Initialization: As many variables as possible should be initialized before program execution.

- PRINT. This idea was used in a larger sense when the Tally Tables were stored to and retrieved from disk files. This reduced the time spent in computing Tally Tables for large surveys from seven hours to two hours and twenty minutes.

Expression Rule 2—Exploit Algebraic Identities: If the evaluation of an expression is costly, replace it by an algebraically equivalent expression that is cheaper to evaluate.

- PRINT. This observation allowed the time of the Tally routine to be reduced from 32.5 seconds per record to 14 seconds per record by realizing that we could delete the operation of adding zero to an integer in the Tally Table.

Expression Rule 3—Common Subexpression Elimination: If the same expression is evaluated twice with none of its variables altered between evaluations, then the second evaluation can be avoided by storing the result of the first and using that in place of the second.

- PRINT. Part of the twenty-minute speedup in the Convert routine was realized by storing the value of `N-100`.

Expression Rule 4—Pairing Computation: If two similar expressions are frequently evaluated together, then we should make a new procedure that evaluates them as a pair.

- PRINT. The Convert routine was structured to exploit the fact that two integers were unpacked from the same byte.

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: CMU-CS-83-108	2. GOVT ACCESSION NO. AD-A127 572	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): A CASE STUDY IN WRITING EFFICIENT PROGRAMS	5. TYPE OF REPORT & PERIOD COVERED: Interim
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): JON L. BENTLEY	8. CONTRACT OR GRANT NUMBER(s): N00014-76-C0370
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA. 15213	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Office of Naval Research, Arlington, VA 22217	12. REPORT DATE: Jan 25, 1983
	13. NUMBER OF PAGES: 37
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for Public release; Distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Approved for public release; distribution unlimited

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)